Interacting with Large Distributed Datasets using Sketch

Eurographics Symposium on Parallel Graphics and Visualization (2016)
W. Bethel, E. Gobbetti (Editors)

Interacting with Large Distributed Datasets Using *Sketch*

Mihai Budiu[1*] Rebecca Isaacs[2*] Derek Murray[2*] Gordon Plotkin[3*] Paul Barham[2*]

Samer Al-Kiswany[4*] Yazan Boshmaf[5*] Qingzhou Luo[2*] Alexandr Andoni[6*]

[1] VMware Research [2] Google Inc. [3] University of Edinburgh [4] University of Waterloo
[5] Qatar Computing Research Institute [6] Columbia University [*] Work performed at Microsoft Research

Abstract

We present Sketch*, a library and a distributed runtime for building interactive tools for exploring large datasets, distributed across multiple machines. We have built several sophisticated applications using this framework; in this paper we describe a billion-row spreadsheet, and a distributed-systems performance analyzer.* Sketch *applications allow interactive and responsive exploration of complex distributed datasets, scaling effectively to take advantage of large computational resources.*

1. Introduction

We target interactive data visualization applications for large datasets. We define a dataset as being "large" if — given the resources of a single machine — manipulating the dataset exceeds an allocated time budget. Visualizing large datasets presents several difficulties: (1) View renderings are limited by screen resolution and human perception — especially a challenge when the number of data points vastly exceeds the number of pixels on the screen. (2) Renderings must be computed at interactive speeds (on the order of seconds). (3) The graphical user interface must enable interaction with the data through the displayed renderings (e.g., zooming, scrolling, filtering, searching, and data transformations); these interactions lead to new data views which must be computed at interactive speeds.

Building interactive visualization applications to satisfy these requirements is difficult. *Sketch* is a library including a distributed runtime that can be used to simplify the construction of such applications. Ideally, *Sketch* should be used as illustrated in Figure 1: when inserted as a drop-in between the data model and the view, it should automatically transform a single-machine application into a distributed application.

We describe in Section 4 two applications that we have built or refactored to run on top of *Sketch*: (1) a spreadsheet for billion-row data sets and, (2) ViewCluster, a distributed

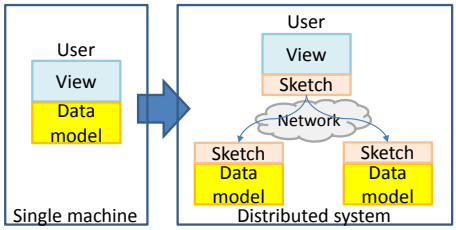

Figure 1: *Application parallelization using* Sketch*: the* Sketch *runtime is inserted between the model and the view, making an application distributed and scalable.*

systems performance analyzer. In practice applications need to be refactored to use *Sketch*; however, we have found this effort to be reasonable (1 man-week for ViewCluster).

Sketch's design is based on two fundamental **principles**:

- Visualizations always display *bounded* views of the input data. Since we cannot *show* all the data, we should *not even compute* views that exceed the display resolution.
- Aggregations are performed using *streaming algorithms*. Streaming algorithms are an active branch of research in big data analysis [Mut05, CGHJ11]; their main feature is using memory sublinear in the size of the input data. In this paper we use the term "*sketching algorithms*" for a particular class of randomized streaming algorithms which: (a) can perform multiple passes over the data (subsequent passes may depend on results computed by earlier passes), (b) use a sublinear amount of memory (usually logarithmic), and (c) produce results insensitive to the input order (i.e., two permutations of an input dataset produce the same result). In this work all aggregations are performed using sketching algorithms.

We exploit two core features of sketching algorithms:

1. Sketching algorithms are naturally parallelizable by construction, since the algorithms can be run concurrently and independently on partitions of the analyzed data, and the final result can be produced by combining the partial results. This enables us to "throw more cores" at a problem.
2. The sub-linear memory usage required by sketching algorithms implies that the partial results are "small"; the partial results are the only input-dependent information that crosses the network. In consequence, network messages are infrequent and small, which is a prerequisite for providing interactive response.

1.1. Visualizations as Sketches

The *Sketch* library exposes remote data to the client-side visualization application using a distributed object abstraction

called a Partitioned Data Set (*PDS*), described in detail in Section 3. *PDS*s expose a very narrow interface; essentially, the only operation for extracting "useful" data that from a *PDS* is to use it the as input of a sketching algorithm (performing some form of aggregation).

One could also implement sketching algorithms on top of a general-purpose traditional distributed analytics engine, such as MapReduce, DryadLinq, or Spark. However, a generic engine is too powerful for our needs: using it one can also write lots of algorithms that can never run at interactive speeds. *Sketch* is a highly specialized distributed analytics engine; it does not support general-purpose queries, but it is optimized especially for executing sketching algorithms. We are thus trading-off generality for speed, by making it difficult for users to use expensive algorithms (e.g., joins).

In general, sketching algorithms compute approximations of the desired results, usually trading off memory or time against precision. Typically the approximation degree is chosen by heuristics. Instead, one important insight we provide is to *derive the approximation degree from the display resolution in order to compute approximate results which are indistinguishable from the exact results*. For example, to draw a histogram the sampling rate should be chosen to make error bars smaller than one pixel in size. One should be unable to distinguish visually between two histograms renderings: one computed using the complete data set and one computed on a sampled dataset.

1.2. Contributions

Sketch borrows many principles from other published systems (see the related work in Section 6). While we do not claim that *Sketch* and our applications are better than competing systems in all respects, we believe the following to be original contributions of this work:

- We describe the *PDS* software abstraction, which allows the manipulation of large distributed data sets through a simple API. Unlike most other big data frameworks (e.g. MapReduce, Hadoop, DryadLINQ [YIF*08], Spark [ZCF*10]), the *PDS* abstraction is provided as **a stateful, distributed, gargage-collected service** — see Section 3. Despite the simple API, *PDS*s do not preclude manipulating complex data models, much richer than a simple relational or nested-relational model — for an example see the ViewCluster application.
- We describe a modular implementation of the *PDS* API. The implementation relies on multiple datatypes *all implementing the exact same interface*. For example, there are separate implementations to contain data, encapsulate parallelism (at the rack, cluster, or core level), implement interprocess communication, provide fault-tolerance, or bound the response time. A distributed system is built by mixing and matching the desired datatypes in complex hierarchies.
- We demonstrate that these abstractions are powerful building blocks for constructing significant visualization tools, describing *two* complex applications in Section 4. We evaluate the scalability of one of the applications on a computer cluster (Section 5). For example, our spreadsheet application computes a histogram of a dataset with 4.1 billion rows in 1.2 seconds seconds using all 1240 cores.

2. Linear Transformations

The *PDS* design is based on a theory of linear transformations. Here we provide the intuitions behind this theory, which is described formally in a separate paper [BP14]. This theory has informed the design and implementation of *Sketch*. In particular, the formalism enables us to give a clear semantics to Sketch computations. Prior work [XZZ*14] has shown that it is easy to abuse parallel frameworks such as Map-Reduce to write programs that are subtly flawed; we also argue in Section 6 that other aggregation network-based systems widely deployed have similar subtle shortcomings. A formally sound design avoids such mistakes. This approach is also espoused by Algebird [Bes14].

Collections and monoids. The core mathematical structure we rely on is a commutative monoid M: a set with a commutative and associative operation $+_M : M \times M \to M$ and a zero 0_M which is the identity element for $+_M$. A typical example of such a monoid is the set of real numbers. In our framework all sketch computations (and, thus, all renderings) are computed on monoids.

A collection can be modeled as an unordered multiset of values (a value can appear multiple times in a multiset). If X is the type of values, we write $\mathscr{C}(X)$ for the type of collections with values of type X. $\mathscr{C}(X)$ is a monoid with multiset union, zero being the empty set.

Given a monoid M, we denote with $M[K]$ the type of key-value dictionaries with keys of type K and values of type M. (If a key is "missing" in the dictionary we define it to map to 0_M). The notation $\{k_0 \mapsto m_0, k_1 \mapsto m_1\}$ represents the dictionary mapping k_0 to m_0 and k_1 to m_1. Dictionaries also form a monoid: given two dictionaries $d, e \in M[K]$, we define $(d+e)[k] =_{\text{def}} d[k] + e[k]$. In *Sketch* dictionary keys may be used for data *location*. For example, a distributed collection of values of type X with partitions on hosts h_0 and h_1 is modeled as a dictionary $d \in \mathscr{C}(X)[K]$; $d[h_0]$ is the partition containing data present on host h_0.

Linear functions. A linear (homomorphic) function between two monoids $f : M \to N$ is a function that "preserves" operations: $f(a +_M b) = f(a) +_N f(b)$, and $f(0_M) = 0_N$. Linear functions are "parallelizable": they can be applied separately on "pieces" and the results can be combined together. The familiar Map and Reduce operators are linear (Map is between two collections, and Reduce between a collection and a monoid of "reduced" values). As we show in Section 3, all operations on *PDS*s are linear.

3. Distributed Datasets

The core abstraction in the *Sketch* framework is a Partitioned Data Set (*PDS*). By packaging data in a *PDS*, user applications manipulate large distributed datasets using multiple machines/cores, without worrying about writing concurrent

```
interface IPDS<T> {
  IPDS<S>          Map<S>    (IMap<T,S> map);
  R                Sketch<R>(ISketch<T,R> s);
  IPDS<Pair<T,S>> Zip<S>    (IPDS<S> other);}
interface IMap<T,S> { S Map(T input); }
interface ISketch<T,R> {
  R                Create (T data);
  R                Combine(List<R> skts);}
```

Figure 2: PDS *C# interface.*

code or handling failures that occur in a distributed setting.

3.1. The *PDS* API

A *PDS* is a generic, distributed, partitioned object. A *PDS* object implements the `IPDS<T>` interface, from Figure 2. (We have slightly simplified the actual interfaces for pedagogical purposes; the actual API also supports asynchronous execution, error and progress reporting, remote background operation cancellation, and other practical features.)

We discuss several implementations of this interface in Section 3.2. One should think of a `IPDS<T>` object as a distributed tree rooted at the client machine and holding a value of type `T` in each leaf (see Figures 3 and 4 for examples). The tree edges are labeled with location information. For example, a path through the tree could be "$\text{rack}_0.\text{machine}_6.\text{core}_4$". Using the formalism from Section 2, a tree of type `IPDS<T>` is a dictionary of type `T`[L*], where L* is the set of *paths*.

The `IPDS<T>` interface comprises only three methods; two of these methods have as arguments user-defined computations (`IMap` and `ISketch`) encapsulated in *closure objects*. Figure 3 illustrates the effects of these methods. These closures and the sketch result of type `R` must be serializable for transport on the network. The types `T` and `S` do *not* need to be serializable; the only data that traverses the network is the result of sketches, `R`.

Map transforms an `IPDS<T>` tree into an `IPDS<S>` with the same "shape" applying the `IMap` function to each leaf.

Sketch runs a sketching algorithm computing a "small" result of type `R`. A sketch object `s` of type `ISketch<T,R>` contains two user defined-functions: $\texttt{s.Create} : \texttt{T} \to \texttt{R}$ and $\texttt{s.Combine} : \mathscr{C}(\texttt{R}) \to \texttt{R}$. Given a dataset dictionary $\texttt{d} = \{l_0 \mapsto v_0, \ldots, l_n \mapsto v_n\}$, we define $\texttt{d.Sketch(s)} = \texttt{s.Combine(s.Create}(v_0)\texttt{,...,s.Create}(v_n)\texttt{)}$. (Note that the paths through the tree are ignored.) The datatype `R` together with the binary operation induced by `Combine` must be a monoid.

Zip combines two dictionaries together into a dictionary of cross-products, pairing values with identical keys. Given two dictionaries `d`, `e`, the result `z = d.Zip(e)` is a dictionary `z` such that `z`[p] = Pair(`d`[p], `e`[p]) for all paths p.

These operations (`Map`, `Sketch`, and `Zip`) are functional: they create a new result, and have no side effects (the user-defined functions `Map`, `Create`, `Combine` supplied are required to have no side effects). They are also linear, as defined in Section 2 (`Zip` is linear in both arguments).

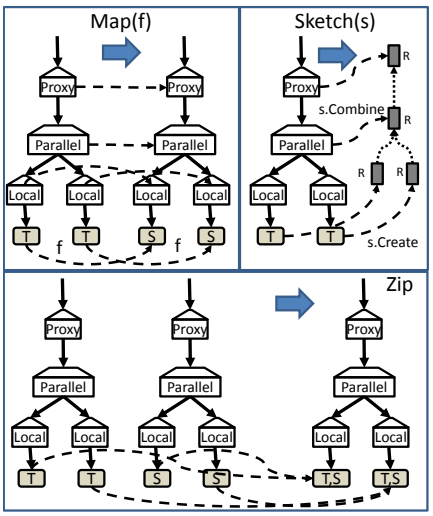

Figure 3: *Computations on* PDS*s.* `Map` *applies a tranformation to each leaf,* `Sketch` *computes a sketch, and* `Zip` *combines two datasets into a single one.*

An `IPDS` is a an opaque remote object that contains some data. That data is not accessible to the client in any way. The client can only manipulate the `IPDS` through its API: `Map`, `Zip`, and `Sketch`. The first two operations create new *PDS*s, to they cannot be used to "extract" any useful data. Unlike MapReduce collections, DryadLINQ `IQueryable` objects, or Spark RDDs, an `IPDS` does *not* provide an iterator interface, so one cannot enumerate the data in a *PDS*s. Thus executing a `Sketch` is the only way to obtain information from a *PDS*. The *PDS* thus forces programmers to think of all data extraction operations as aggregations.

3.2. Dataset Implementations

*PDS*s are abstract objects, accessed solely through the `IPDS` interface. *Sketch* provides several basic implementations of this interface. Each implementation solves a particular problem arising when building a distributed system. Because all component pieces have the same interface they can be stacked in arbitrary ways and so by mixing and matching these implementations, users of the *Sketch* framework can build distributed *PDS*s with various degrees of parallelism, concurrency, network transparency, and resilience. This modular construction of a distributed system is powerful, elegant, and is an original contribution of this work.

- `LocalPDS<T>` is a container for one value of type `T`.
- `ParallelPDS<T>` contains a list of child `IPDS<T>` objects. All processing on these objects is performed independently and in parallel. Running a `Map` on a `ParallelPDS` returns a new `ParallelPDS`. `Sketch` runs recursively on the children and then uses `Combine` to assemble the partial results produced into the final one.
- `ProxyPDS<T>` contains a single child `IPDS<T>` located on a remote machine. The proxy uses a remote method invocations to forward `Map`, `Sketch`, and `Zip` calls to

the remote *PDS*. The proxy also relays back to the client results and exceptions that occur during remote method execution. `ProxyPDS` also participate in the distributed garbage-collection of *PDS*s (more details about the distributed garbage-collection are presented in the companion technical report [BIM*15]).

• `LazyPDS<T>` contains a single child of type `IPDS<S>`, and a "frozen" closure `map` of type `IMap<S,T>`. For a lazy dataset `l`, `l.Map(t) = LazyPDS(l.child,Compose(l.map, t))`. I.e., the lazy dataset postpones the execution of any `Map` by composing closures and returns immediately.

Invoking a sketch `s` on a lazy dataset `l` creates a new sketch `sl` by composing `s` with the frozen map: `sl = Compose(l.map,s)` and invokes it `l.child.Sketch(sl)`.

• `ReplicatedPDS<T>` is similar to a parallel dataset, but all its children must represent *identical values*, i.e., they are replicas of one dataset, ideally residing in independent fault domains. Like parallel datasets, replicated datasets forward calls to all their children. However, their behavior differs in the presence of errors (exceptions): if an exception occurs while running on a child, the replicated dataset swallows the exception and marks the corresponding child as unavailable. Exceptions that occur on all children are considered deterministic and are returned to the caller. `ReplicatedPDS` thus provides fault-tolerance.

• `HedgedPDS<T>` is a cross between a `PartialPDS` and a `ReplicatedPDS`, which implements hedged requests: a request is sent to all its identical children, and the first response received is immediately relayed back [DB13]. `HedgedPDS` are instrumental in reducing significantly the tail response time of requests, especially when dealing with a large number of machines.

Sketch relies on a custom Remote Method Invocation implementation, built on top of Windows Communication Foundation. The RMI layer provides the support needed for distributed garbage-collection, caching and fault-tolerance; *leases* are used to prevent client crashes from leaking server resources. More details about the RMI layer are available in a companion technical report [BIM*15].

3.3. Putting the Pieces Together

Figure 4 shows a typical tree-shaped distributed *PDS* object. The leaves of the tree are all `LocalPDS` objects located on the server machines, holding references to the partitions of the data (of type `T`) that is being processed. On each server, all leaves have a common `ParallelPDS` parent; this parent effectively utilizes all cores on each server in parallel.

Server 1 contains a second `ParallelPDS`, with two children, both in Rack 0 (one being on the same machine); the child in Server 0 is referenced through a `ProxyPDS`. This second-level `ParallelPDS` provides rack-level aggregation, effectivelly utilizing the higher bandwidth available in a rack to the top-of-the-rack switch.

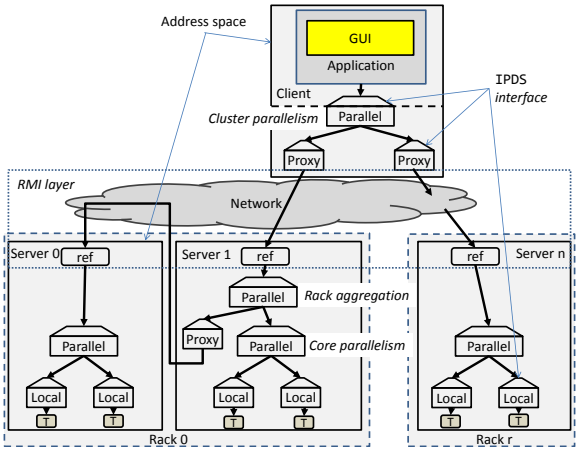

Figure 4: *Sample distributed* PDS *state spanning multiple address spaces on multiple machines.*

Finally, the client machine contains a set of `ProxyPDS` onbjects, one for each rack. These are all children of a root `ParallelPDS`. The root invokes computations concurrently on all children, providing cluster-level parallelism.

The visualization application interacts with the root of this distributed tree as with an object in its local address space. In fact, the client cannot tell the difference between a `LocalPDS` running on the local machine and a distributed data set running on the cluster. Even failures on remote machines (including machine crashes) are handled transparently by the *PDS* abstraction. Errors that occur on remote machines can be either automcatically handled (e.g., by `ReplicatedPDS` or `HedgedPDS` objects), or are are exposed as standard C# exception, which are usually trapped by the GUI and displayed as error messages.

The *Sketch* service does not have a central coordinator; each server provides the service completely independently of its neighbors. Each client acts as a separate control point for the computations it initiates. We have not explored the interference caused by running multiple clients simultaneously on the same service.

To provide fault tolerance, the input data must be replicated (replication is performed by the storage layer, e.g., a filesystem such as the Cosmos distributed filesystem [CJrL*08]). To handle a replicated dataset the client inserts `ReplicatedPDS` nodes when loading the data. Interestingly, `ReplicatedPDS` nodes can be used at any level in the *PDS* tree: either just above leaves, above the rack-level nodes, or even as the root of the complete hierarchy.

Figure 3 shows how computations operate on *PDS*s. Running a `Map` on a *PDS* produces another *PDS* with the exact same shape and with the corresponding components located in the same address spaces as the sources. Running a `Sketch` produces a scalar result, that aggregates data from leaves up the tree. Executing a `Zip` on two *PDS*s with the same shape produces as a result with the same shape. In consequence, all *PDS* that descend from the same source have

the exact same shape.

For the client all method invocations on a distributed dataset are single-threaded and atomic, and seem to be executed in the local address space; in reality all computations run concurrently in a fault-prone distributed system.

Every computation on a *PDS* may involve thousands of concurrent threads on multiple machines; however, all computations on a *PDS* behave as simple atomic operations that complete when a result is returned to the client. No combination of system errors or network partitions can create or leak an incorrect or partial *PDS* object.

4. Applications

In this section we describe two complex applications that were built on top of the *Sketch* framework: we start with a distributed spreadsheet, and then we discuss a distributed system performance monitoring application. Both applications render data views bounded by screen resolution — independent of the size of the displayed data.

4.1. A Spreadsheet for Large Data

We describe a spreadsheet for browsing distributed datasets. Figures 5 and 6 displays several screenshots showing different views of a dataset comprising 4.1 billion tweets with 12 columns. Users can filter, search, compute new columns, sample, find heavy hitters, sort on arbitrary column combinations, scroll, draw charts (multidimensional histograms and heatmaps), and perform set operations between two views of the same dataset (intersection, union, difference, etc.). A real-time video of the spreadsheet in action is available at `https://youtu.be/NkV5r7OzCoc`. We use references to the video: **V**(1:23) represents the video at time 1:23. A brief user manual is available as [Bud14].

Building interactive applications for this data size is very challenging, but the *Sketch* framework automates most of the work required for exploiting the resources of a computer cluster and hiding network communication. The challenge is to express all of the above operations in terms of the *Sketch* API from Figure 2.

The spreadsheet operates on database-like tables, with a known schema (i.e., a list of typed columns). A table can have few columns (hundreds) and many rows (billions); the rows are partitioned on machines and cores. The table data is represented as an `IPDS<Table>` object, where `Table` is a simple (non distributed), immutable, relational-like table object. The `Table` implementation has a schema, an array of columns, and a bitmap with 1 bit per row, used for filtering.

The spreadsheet application does not depend on the actual storage substrate. Our implementation can ingest data from files, SQL Server databases (one for each server), the Cosmos [CJrL*08] distributed storage system (similar to the Hadoop filesystem), and from a custom distributed column-store. The `Table` lazily loads the browsed columns in RAM when using the column store (one `Table` per core).

Each spreadsheet window maintains a reference to an `IPDS<Table>` and shows a *rendering* of the associated

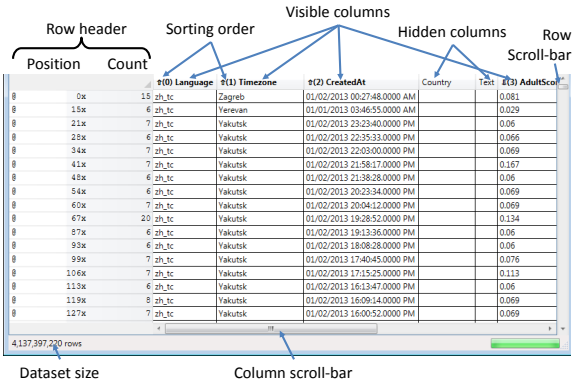

Figure 5: *Tabular representation of a large dataset with 4.1 billion rows and 12 columns. There are 17 visible rows representing 134 data rows.*

data view. User actions can either change the rendering (e.g., scroll, chart), or change the dataset itself (e.g., filter), causing a recomputation of the rendering. *All* renderings are computed using sketches.

4.1.1. Data views

The main challenge of a billion-row spreadsheet is to fit as much information into the very limited screen real-estate. The spreadsheet can display three types of data views: tables, histograms (and arrays of histograms) and heatmaps.

Tabular views

We have explored several alternatives for displaying data in a tabular fashion, before settling on the solution we describe here. A typical screen can accomodate hundreds of rows of tabular display, out of billions. The core insight is to represent a *data cube* in each row.

In general, only some columns are visible (the user can choose to hide or display columns at any time). The displayed data is *always* sorted (in some lexicographic order) on *all* visible columns; in Figure 5 the data is sorted descending on the Language column, descending on Timezone, Descending on CreatedAt and ascending on AdultScore. Each displayed row corresponds to a data cube computed from the dataset projected on the displayed columns. The row header displays both the position of the row in the sorted order, and the size of the cube (i.e., the number of occurrences of the respective projected row in the full dataset). For example, the first row in the table corresponds to 15 data rows whose projection on the 4 visible columns is the one displayed. See also the accompanying video for how cubes are re-computed as additional columns are un-hidden **V**(1:36). The cube display is very compact when there are few distinct values in the visible columns. Computing these cubes is very fast, because only the visible cubes need to be computed – as described in the next section. (Note that the cubes are sorted on the displayed columns, and not on the cube size.)

The vertical (row) scroll-bar is a very important GUI element. The vertical position of the scroll-bar represents the

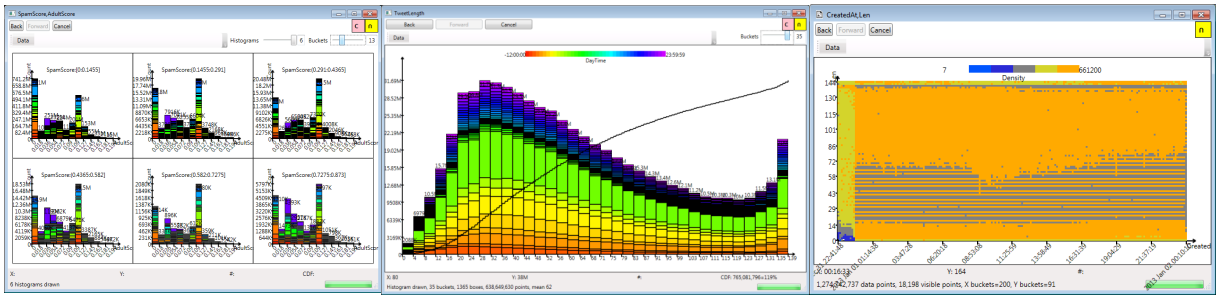

Figure 6: *Spreadsheet screenshots of a dataset of tweets with 4.1 billion rows and 12 columns, manipulated on a cluster with 155 machines (1240 cores): 3D histogram (array of histograms): showing SpamScore/AdultScore/TweetLength, 2D histogram with CDF showing tweet length distribution colored with tweet time, and heatmap of tweet length vs creation time.*

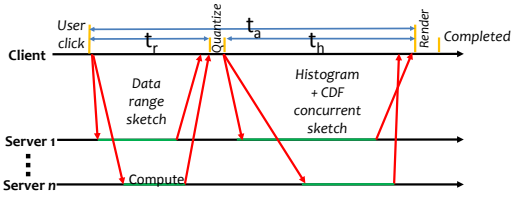

Figure 7: *Execution timeline for computing a histogram; diagonal arrows show messages between client and servers.*

position of the first displayed row in the sorted order. The size of the scroll-bar represents the amount of data that is visible on the screen. For Figure 5 the window contains 134 data rows, so the scroll-bar height is $WH \times 134/(4.1 \times 10^9)$, where WH is the window height. This implies that the size of the vertical scroll-bar can change when scrolling, based on the distribution of the visible cubes within the dataset; for example, if a displayed row accounts for half of the data, the scroll-bar would be at least half of the window size. While this visual effect is unusual, we found that it conveys useful additional information about the data distribution.

Histograms and heatmaps

Charting large datasets requires aggregating information. The *Sketch* spreadsheet enables visualizations using multi-dimensional histograms **V**(1:17) and heatmaps **V**(2:19). As noted in [IVM13, LJH13], when displaying very large datasets, with many more points than pixels on the screen, charts *must* show aggregate data. The *Sketch* spreadsheet supports the chart types suggested by the cited prior work; all these charts are essentially multidimensional histograms.

Using the formalism in Section 2, a histogram is a dictionary value of type $\mathbf{R}[B]$, where B is the set of buckets and $\mathbf{R}$ is the set of real numbers; if we fix a set of buckets B, histograms over those buckets form a monoid. Heatmaps are dense 2D histograms.

As shown in Figure 7, a histogram is computed in 3 steps: (1) a Sketch to *compute the range* of the data (minimum and maximum values), (2) a client-side quantization computation to determine the buckets B, and (3) a second Sketch to compute the histograms proper. The number of buckets $|B|$ is always chosen to be smaller than the number of pixels on the X axis. While the X axis resolution decides the number of buckets, the Y axis resolution decides the sampling rate: which is chosen so that the error bars are smaller than 1 pixel (with high probability, in the worst case, for all possible data distributions). In general the data range cannot be computed on sampled data, since it may lose rare outliers.

Figure 6 middle shows a two-dimensional histogram (where each bar is a stack) **V**(1:25). The image also displays the a line plot of the cumulative distribution function (CDF). Computing naively the CDF would require sorting the complete dataset; however, the CDF can be computed very efficiently by numerically integrating a histogram where each bucket has the width of exactly 1 screen pixel.

4.1.2. Interacting with data

Each window maintains a *history* of the rendered data together with the renderings themselves, enables the user to navigate back and forth instantly. (The history and renderings are all small objects, which fit in the client memory.)

Sorting is the main operation performed on tabular data displays. Sorting is very fast; for example, it takes less than 1/2 second to sort through 4 billion rows using 155 machines when data is already loaded from disk **V**(1:37).

The secret is only to sort the rows that can must be shown to the user. Sorting is implemented using a generalized "`NextK`" `Sketch`. The `NextK` sketch has the following parameters:

- The lexicographic sort order to use: this is the same as the set of "visible" columns
- f: the (projection on the visible columns) of the first row to display on the screen (f is used for scrolling within a sorted dataset)
- k: the number of rows to sort

`NextK` computes a dictionary containing the cubes to display on the screen; each key of the dictionary is a row of the dataset (projected on the visible columns) and the corresponding value is number of times the row occurs in the full dataset (shown in the row header count, in Figure 5). The algorithm for computing `NextK` is very simple:

- Tuples smaller than f in the indicated lexicographic order are discarded.

• From the remaining tuples the k smallest are kept, maintaining the number of occurrences of each. (If there are fewer than k distinct tuples, all of them are kept.)

The set of k smallest tuples with described operations forms indeed a commutative monoid ("zero" is a set with no tuples), so `NextK` is a proper Sketch. The computation can be performed very efficiently using a priority queue; for a balanced dataset the complexity is $O(N/C \log k)$, where N is the dataset size, and C is the number of cores.

Scrolling a sorted dataset is performed at interactive speeds. No other visualization or analytics engine that we know of provides this capability.

There are two types of scrolling:

1. **Paging** (up/down) is very efficient. Scrolling down is performed using a `NextK` sketch with f set to the last visible tuple, and scrolling up is a `NextK` sketch with f set to the first visible tuple and the sort direction reversed.
2. **Dragging** the scroll-bar is the most complex operation on a tabular view. This is done in two stages: determining the new "first tuple" f, and then running `NextK`. To compute efficiently f we again take advantage of the limited screen resolution: since each pixel displacement of the scroll-bar can represent many millions of rows in a large dataset, we can choose *any* of these rows as the starting tuple. The scroll-bar position is in fact representing a *quantile*: for example, in a window 200 pixels tall, the scroll-bar position 3/4 from the top represents some percentile between 75% and 75.5% (since each pixel represents 0.5% of the data). We use a well known sketching algorithm for computing such an approximate quantile (e.g. [CM05]), with cost logarithmic in the required precision.

Computing new columns: the user can type an arbirary C# function to compute values for a new column. The input to the function is a data row; the function is applied to each row separately **V**(3:26).

Filtering can be performed similarly, by evaluating an arbitrary predicate on each data row. Filtering uses bitmaps to represent efficiently subsets of a large dataset. Sampling is a special form of filtering.

Combining two datasets: Each view of the dataset has a little yellow icon on the top-right displaying a "set intersection" sign. By dragging and dropping that icon **V**(1:00) from a window displaying a source dataset S to a window representing destination dataset D (over the same underlying table), the user is presented with a pop-up menu with a set of choices for a set operation op: union, intersection, difference, replacement, etc. The result of this manipulation is to perform $D = D\ op\ S$, and the view of the destination's window is recomputed. This kind of direct manipulation for large datasets is described in detail in a separate technical report [Bud10], and is remarkably effective for navigating multiple views of the same underlying dataset.

Heavy hitters is a computation that can find the most frequent tuples (or tuple projections) in a dataset; it can be computed using the standard Misra-Gries sketching algorithm [MG82]. The user inputs a *frequency*; the algorithm is guaranteed to produce all tuples that occur with at least the specified frequency.

Most charts interaction is via direct manipulation:

Zooming in is performed by dragging a box with the mouse; the box determines a predicate, that is used for filtering **V**(4:13). Zooming can be used both to exclude data outside of the box, or the data inside the box. Zooming into a histogram may cause the buckets to be "blown up," revealing the finer-grain substructure of the data.

Pivoting is also performed as described in [Bud10]: by performing drag-and-drop with the little pink C icon between two displayed histograms **V**(4:58). The "C" stands for "color"; the effect is to divide each bucket in the destination histogram into colored sub-buckets that correspond to the bars in the source histogram, as shown in Figure 6 (in the Figure each bar represents a tweet length; the colors represent the time-of-day when each tweet was issued). Pivots are quite effective for visually detecting (partial) correlations between the two columns that are charted **V**(5:52).

4.2. ViewCluster: Distributed System Perf. Analysis

In this section we describe briefly the ViewCluster tool, which is used for offline performance analysis of distributed systems by visualizing distributed systems event traces. Figure 8 shows 2 screenshots of ViewCluster.

An ideal tool to understand and debug the performance of a distributed system allows the user to visualize performance at multiple resolutions (e.g., cluster/machine/thread and second/millisecond/microsecond). ViewCluster collects a large variety of low-level events (context switches, inter-thread signals, deferred procedure calls, interrupts, network packet sends and receives), and high-level activity metrics (memory allocations, garbage collection, and application-specific events). Fine-grained event tracing produces tens of thousands of events/machine/minute, amounting to several gigabytes/minute/cluster. Note that the data model is significantly more complex than just a relational table. ViewCluster is a proof that the *Sketch* framework can be used to visualize dataset with complex structures (object graphs), and not just tabular data.

The original version of the ViewCluster application predated the *Sketch* framework, and was a single-machine implementation; it copied traces from all analyzed machines to the client machine for analysis. By refactoring the tool around the *Sketch* framework we have obtained a distributed application which browses and analyses the traces in place, on the machines where they are collected; all machines in the analyzed cluster can participate in the trace browsing and visualization. The porting effort took around one person-week.

Using *Sketch* provides immediate benefits: (1) data is never moved from the original machines; (2) the work performed by each machine is constant, irrespective of the size of the analyzed system: trace parsing and data navigation are

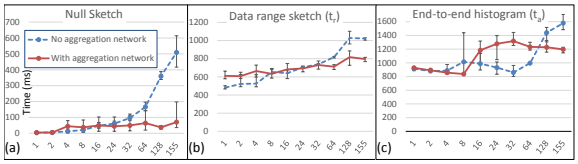

Figure 9: *Scale-out experiment, where we keep a constant amount of data per server and increase the number of servers. Graph displays end-to-end time measured at client for computing various sketches. X axis is number of server machines, Y axis is time in milliseconds.*

performed in parallel on all machines, and scale with the size of the monitored system.

Visualization is a powerful tool for performance debugging, but providing a good user experience requires interactive response times. Data loading is a relatively slow process (linear in the size of the collected traces, one the order of minutes); loading requires parsing on-disk trace files to construct a complex in-memory representation of system activity on each server, including multiple indexes used to navigate the data structures quickly on user requests. After parsing, data exploration can be done at interactive speeds: the response time is on the order of seconds for clusters with tens of machines (which involves browsing sets of tens of millions of events). The user can scroll on both axes, and zoom in/out; she can also change visualization details or can mouse-over to find out more details about a specific event.

All renderings, as shown in Figure 8, are computed using sketches. Each sketch is parameterized by the time range visible on the screen and by the window resolution in pixels. For example, in the left view, average CPU utilization is computed and displayed for each time interval corresponding to 5 pixels, which is about .5ms in this image. Zooming requires filtering and recomputing aggregations at finer time-scales. The server-side computations take advantage of the index data structures constructed during trace parsing to touch only those events visible in the displayed rendering.

5. Evaluation

Here we evaluate properties of the *Sketch* framework using the Spreadsheet application. We measure first interactivity and scalability (Section 5.1), next we compare performance with VisReduce [IVM13] using a common dataset (Section 5.2).

Hardware platform. We run our experiments on a 155 node cluster. All machines use Windows Server 2012 R2. Each is equipped with an 8-core 2.1GHz AMD Opteron 2373 EE processor, and at 16GB of RAM. The machines are organized in 5 racks, each with a top-of-rack switch. The network is 1Gbps Ethernet. In our evaluation there is only one client connected to the service, on the same LAN.

5.1. Interactivity and Scalability Evaluation

In the first set of experiments each server processes the same amount of data. The dataset is a table with 12 columns representing a set of tweets; the full dataset, shown in Figure 6, contains 4.1 billion tweets. Total data size is about 1TB. Data is partitioned almost uniformly on machines, for 26.45 million rows per machine, further divided into 8 parts on each machine (one part per core), of 3.3 million rows in each part. The underlying storage is a simple custom column store that we have implemented.

Browsing speed. We measured the end-to-end execution time for each operation for an interactive user-controlled data-browsing session on the full 4.1B rows dataset (histograms, tabular views with scrolling, and heatmaps). The average latency between the user initiation and the final rendering is 560ms; with a maximum of 7.6s. Loading data is the most expensive step, in particular non-compressible string columns require expensive I/O.

The remaining experiments in this section were scripted (there was no user think time). Each experiment is repeated 7 times, and error bars show variability after the two extreme values are dropped. The file caches are warmed, so disk I/O cost is not a factor in these results. Result memoization is disabled, because it would provide instantaneous answers when running repeated computations.

Communication latency. To get a baseline for the communication costs we measure the end-to-end execution time for a Null sketch; the Null sketch `Create` and `Combine` function just return `true`. Figure 9(a) shows 2 sets of measurements: with (red/continuous line) and without (blue/-dashed line) per-rack aggregation.

Without rack-level aggregation, the Null sketch latency increases linearly with the number of servers, as expected, due to contention on the single client network interface communicating with all servers. The rack-level aggregation layer provides a flat response time over this set of machines. The two lines cross at 16 servers. With similar constants we expect that a second-level intermediate aggregation layer is useful for clusters with more than 16*16 = 256 machines. Facebook's Scuba [AAB*13] uses an aggregation fanout of 5, which is much smaller.

Scale-out. The next experiment measures the time to compute the histogram of the CreatedAt column, which contains the DateTime when each tweet was created. Figure 7 shows the timeline of the computation when there is no rack-level aggregation. Displaying a histogram requires 3 sketches, as explained in Section 4.1. (1) The first computes the range and precision of the data. After receiving these results, the client decides the histogram bucket boundaries and initiates 2 more sketches concurrently: (2) the histogram proper, and (3) the CDF. After receiving these results the client renders and displays the histogram on the screen (as in Figure 6 top right).

Figure 9(b) shows the execution time (t_r in Figure 7) of the first sketch, that computes the data range and precision. Figure 9(c) shows the end-to-end time (t_a in Figure 7) to compute all 3 sketches. The data-range computation dominates at 65% of the end-to-end histogram time. The last 2 sketches are computed on sampled data, as explained in Section 4.1.

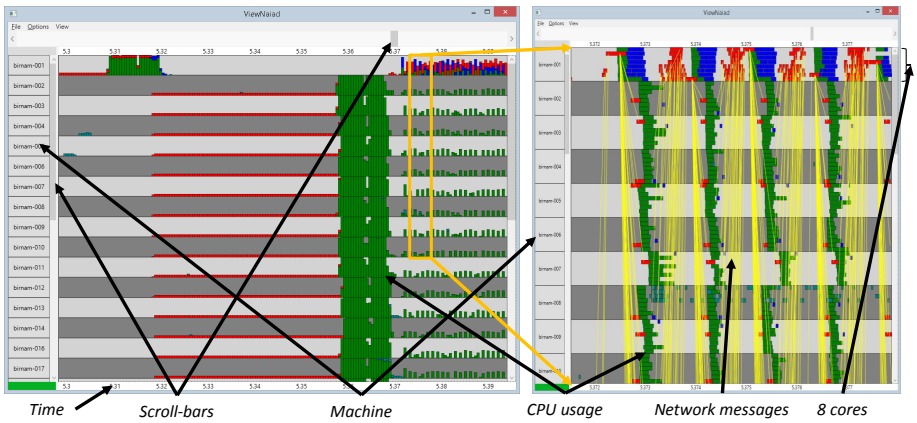

Figure 8: *Two ViewCluster screenshots at different resolutions, while analyzing of a distributed application. The X axis is time, and the Y axis is the machine/core. The left screenshot covers 0.1 seconds and 17 machines, while the right screenshot is a zoomed-in display of the indicated region, covering 6 milliseconds and 10 machines. Each "row" shows the activity on one computer. Colors indicate the type of activity (user-level, system-level, garbage-collection, interrupt handler, user-specified code region, etc.). The left image shows average CPU utilization across all 8 cores using stacked bars in each time interval; when zooming in each core is presented as a separate horizontal bar, and network messages become visible (yellow lines).*

(The data range needs to be recomputed every time filtering operations are applied.)

5.2. Visualizing OnTime Flight data

In order to provide a comparison with related work, we reused the dataset from VisReduce [IVM13]: the OnTime flights database [ont]. The dataset consists of 321 files, each around 250MB, for a total of 65G uncompressed. There are 159M tuples, with 109 columns each. We distribute the files in a round-robin fashion on our machines. We experiment with 5 machines as in [IVM13], with 64 files/machine. We also experiment with 155 machines, with about 2 files/machine (3 files on 11 machines). The files on a machine are all loaded as children of a single `ParallelPDS`.

Since it is difficult to make a direct comparison, we have carried several experiments computing the histogram of column "FlightDate", summarized in the table below.

Experiment	Time (s)
VisReduce [IVM13], 5 servers, cold cache	0.5-5.5
Sketch, 5 servers, cold cache, end-to-end	3.2
Sketch, 5 servers, warm cache, end-to-end	2.5
Sketch, 5 servers, histogram only	0.047
Sketch, 155 servers, warm cache, end-to-end	1.3
Sketch, 155 servers, histogram only	0.09

6. Related Work

Parallel Computation and Visualization. It is challenging to do justice to the immense amount of work on parallel visualization and rendering, starting with [PSGL94]. There are sophisticated parallel rendering and visualization toolkits available, for example ParaView [Squ07]. These tend to have much richer data models and to provide much more sophisticated functionality than Sketch. The *Sketch* design decouples the parallel-execution framework from the actual data model and rendering.

Sketch is a "sort-middle" rendering pipeline [Cro95], because the data is arbitrarily partitioned, and the rendering is computed on the client side from aggregated data summaries. Initial versions of *Sketch* attempted to do rendering sort-last, by overlaying actual bitmaps rendered on a transparent background. The examples in [BCS13] show why such an approach is incorrect: overlaying transparent bitmaps with absolute color values leads to saturated color values, truncating information; fundamentally this happens because bitmap overlays do not form a commutative monoid. Twitter's Algebird [Bes14] open-source library approaches the task of (non-interactive) data analytics using algorithmic tools related to our monoids and linear transformations.

Sketch is to some degree related to Zoomable User Interfaces, introduced by the Pad system [PF93]. ZUIs tend to have multiple distinct semantic layers (e.g. [Ham14] and [JE10]). The idea of Responsive Design [Ada04] is to adapt a web site to the screen resolution; *Sketch* extends this idea to data visualization (but *Sketch* is not the first system to propose this approach).

Polaris [STH08], commercialized under the name "Tableau" is a successful visualization platform for tabular data. *Sketch* is a complement to tools such as Excel or Tableau, designed to work in the regime where there are many more data points than pixels on the screen, and data exceeds the resources available to a single machine.

The MPI Reduce [SOHL*96] primitive is strongly related to the aggregation model of *Sketch*, but it is a lower-level, non generic interface, oriented towards numeric data.

Large-scale analytics. The *PDS* `Sketch` operation is functionally equivalent to the Neptune Data Aggregation Call architecture [CTYS03]. Unlike Neptune, *Sketch* provides a stateful object model (the *PDS*) with distributed garbage-collection. Our work is also geared towards visualization applications.

Several systems including Dremel [MGL*10] (aka. BigQuery), PowerDrill [HBB*12], Apache Drill [HN13], Druid [YTL*14], and Scuba [AAB*13] are oriented towards efficient execution of aggregated queries and visualization of large distributed databases. *Sketch* decouples the aggregation layer from the application, and allows manipulation of very rich datatypes and displays. In particular, applications such as ViewCluster show that *Sketch* is applicable to rich data types, beyond simple relational data models. Some of these system do not provide a clear semantics when results are very large (e.g., Scuba limits results to 100K rows to "avoid memory issues and rendering problems", and Dremel computes Top-K only approximately). The ATLAS system [CXGH09] targets the large-scale browsing of time-series data, using prefetching to decrease operation latencies.

Big Data Abstractions. The *PDS* is similar in some respects to a Spark RDD [ZCF*10]; *PDS*s however are garbage-collected and memoized, and they offer a narrower interface, optimized for distributed hierarchical aggregation. It is unclear how easily an application such as our distributed profiling tool ViewCluster, which uses very complex data structures and indexes, can be implemented on top of Spark; we are not aware of any application of similar complexity implemented over a big data framework.

The modular architecture of Anvil [MHK09] is similar to the various implementations of the *PDS* interface in our distributed objects. Several systems are stateful; for example: Grappa [NHM*14] keeps state local and delegates computation to remote locations, but the computational model is quite different; [FMKP14] is geared towards batch processing, and Piccolo [PL10] uses a distributed key-value store.

[EF10] presents a theory of aggregated information visualization. The idea of restricting visualizations to aggregates for large data is proposed by imMens [LJH13]; imMens precomputes cubes for faster rendering. The Nanocubes project [LKS13] adopts this approach for visualizing spatiotemporal data sets. DICE [KJTN14] also tackles interactive cube browsing of large relational datasets prefetching (similar to ATLAS) and sampling are used to decrease user-perceived latencies. VisReduce [IVM13] also uses a system architecture for computing user-defined aggregation functions, as well as incremental renderings. MapReduce [DG04] is adapted for complex big-data rendering in [VBS*11], foregoing interactive response. [BCS13] identifies several problems with traditional visualizations, but proposes heuristic solutions. *Sketch* uses the screen resolution to select rendering precision by restricting errors bars to be below one pixel in size, computes CDFs as histograms with pixel-sized bins, computes NextK based on the number of displayed rows, and scrolls quickly through sorted views using approximate quantiles, all new ideas.

Incremental Visualizations. A fair amount of work focuses on incremental visualizations. Early work [HHW97] synthesizes incremental queries on a database. [FPDS12] evaluates user interfaces for incremental visualizations; Tempe [Mic] uses a generic query language; [CGQ14] uses special user annotations on data; Stat! [BCD*13] uses a programmable streaming engine. The problem of distributed aggregation is one of the core problems of sensor networks [RV06]. *Sketch* is built on top of a distributed, garbage-collected partitioned object model, and is not tied to a database. It would be an interesting exercise to adapt the *Sketch* model to incremental visualizations.

Sampling. There is also a large amount of work on sampling-based data visualization; recent work related to big data is BlinkDB [AMP*13] and [YCZ14]. These techniques could further accelerate the *Sketch* Spreadsheet.

Application Specific Visualization. Splunk [spl] is a log-analysis system for distributed logs, related to our log-browser spplication; it has a similar query architecture, where the "search head" corresponds to our client, and the "indexers (search peers)" correspond to the *Sketch* servers. *Sketch* seems to provide better scalability than Splunk, but could certainly benefit from the rich set of log-parsing heuristics and tools. IBM BigSheets [big14] also provides a spreadsheet user interface for big-data manipulation.

The Fay log-analysis system [UEPPB11] is related to ViewCluster; however, our viewer is optimized for browsing, and not for analytics.

Data virtualization [Stu09] is a well-known technique for building large spreadsheets, and closely related to paginated queries in databases and to the display of results in search engines. The architecture of search engines also relies on aggregation trees. All these influenced the *Sketch* spreadsheet.

7. Conclusions

The core observation driving this work is that visualizations render data with precision necessarily bounded by the screen resolution. To render a large dataset on a small screen one has to aggregate information in some way. We answered the question: "What if aggregation is the *only* operation that can be applied to a large dataset?" (Any sequence of Map, Zip and Sketch operations is equivalent to just a Sketch.)

To answer this question we constructed a generic framework for building distributed aggregation and used it to build two useful and complex data visualization applications: a spreadsheet, and a tool for debugging distributed application performance. The aggregation interface endows these application with desirable properties, such as simple parallelization, low-bandwidth requirements and natural scalability, which we have quantified up to 1240 cores.

We described Partitioned Data Sets, or *PDS*s, a distributed stateful object architecture for building distributed aggregation networks. *PDS* objects hide the complexity of programming distributed systems from clients. An interesting benefit of the narrow *PDS* API is that it *forces* developers to code their application using early data reduction, guiding them towards efficient solutions.

References

[AAB*13] ABRAHAM L., ALLEN J., BARYKIN O., BORKAR V. R., CHOPRA B., GEREA C., MERL D., METZLER J., REISS D., SUBRAMANIAN S., WIENER J. L., ZED O.: Scuba: Diving into data at Facebook. *Proceedings of the VLDB Endowment 6*, 11 (2013), 1057–1067. 8, 10

[Ada04] ADAMS C.: Resolution dependent layout. http://www.themaninblue.com/writing/perspective/2004/09/21/, September 21 2004. 9

[AMP*13] AGARWAL S., MOZAFARI B., PANDA A., MILNER H., MADDEN S., STOICA I.: BlinkDB: Queries with bounded errors and bounded response times on very large data. In *European Conference on Computer Systems (EuroSys)* (Prague, Czech Republic, 2013). URL: http://www.cs.berkeley.edu/~sameerag/blinkdb_eurosys13.pdf. 10

[BCD*13] BARNETT M., CHANDRAMOULI B., DELINE R., DRUCKER S., FISHER D., GOLDSTEIN J., MORRISON P., PLATT J.: Stat!: an interactive analytics environment for big data. In *ACM SIGMOD International conference on Management of data (SIGMOD)* (New York, NY, USA, 2013), pp. 1013–1016. URL: http://doi.acm.org/10.1145/2463676.2463683, doi:10.1145/2463676.2463683. 10

[BCS13] BATTLE L., CHANG R., STONEBRAKER M.: Dynamic reduction of query result sets for interactive visualization. In *IEEE International Conference on Big Data* (Oct 2013), pp. 1–8. doi:http://doi.org/10.1109/BigData.2013.6691708. 9, 10

[Bes14] BESSALAH S.: Algebird: Abstract algebra for big data analytics. Devoxx Conference, November 11 2014. URL: https://speakerdeck.com/samklr/algebird-abstract-algebra-for-big-data-analytics. 2, 9

[big14] IBM Big Sheets. http://www-01.ibm.com/software/ebusiness/jstart/bigsheets/, Retrieved 2014. 10

[BIM*15] BUDIU M., ISAACS R., MURRAY D., PLOTKIN G., BARHAM P., AL-KISWANY S., BOSHMAF Y., LUO Q., ANDONI A.: *Interacting with Large Distributed Datasets Using Sketch*. Tech. Rep. TR1817, University of Wisconsin-Madison, January 2015. URL: http://digital.library.wisc.edu/1793/70467. 4

[BP14] BUDIU M., PLOTKIN G.: Multilinear programming with big data. Festschrift for Luca Cardelli, September 2014. 2

[Bud10] BUDIU M.: *User interfaces for exploring multi-dimensional data sets*. Tech. Rep. MSR-TR-2010-67, Microsoft Research, June 2010. URL: http://research.microsoft.com/apps/pubs/?id=132078. 7

[Bud14] BUDIU M.: Sketch spreadhsheet user guide. http://budiu.info/work/SketchUserManual.docx, April 2014. 5

[CGHJ11] CORMODE G., GAROFALAKIS M., HAAS P. J., JERMAINE C.: Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases 4*, 1 (December 31 2011). URL: http://www.nowpublishers.com/articles/foundations-and-trends-in-databases/DBS-004. 1

[CGQ14] CHANDRAMOULI B., GOLDSTEIN J., QUAMAR A.: Scalable progressive analytics on big data in the cloud. In *International Conference of Very Large Data Bases (VLDB)* (August 2014). URL: http://research.microsoft.com/apps/pubs/default.aspx?id=200169. 10

[CJrL*08] CHAIKEN R., JENKINS B., ÅKE LARSON P., RAMSEY B., SHAKIB D., WEAVER S., ZHOU J.: SCOPE: Easy and efficient parallel processing of massive data sets. In *International Conference of Very Large Data Bases (VLDB)* (2008). URL: http://research.microsoft.com/en-us/um/people/jrzhou/pub/Scope.pdf. 4, 5

[CM05] CORMODE G., MUTHUKRISHNAN S.: An improved data stream summary: The Count-Min sketch and its applications. *J. Algorithms 55*, 1 (Apr. 2005), 58–75. URL: http://dx.doi.org/10.1016/j.jalgor.2003.12.001, doi:10.1016/j.jalgor.2003.12.001. 7

[Cro95] CROCKETT T. W.: Parallel rendering. *Parallel Computing 23* (1995), 335–371. 9

[CTYS03] CHU L., TANG H., YANG T., SHEN K.: Optimizing data aggregation for cluster-based Internet services. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (New York, NY, USA, 2003), pp. 119–130. URL: http://www.cs.rochester.edu/u/kshen/papers/ppopp2003.pdf, doi:http://doi.acm.org/10.1145/781498.781517. 9

[CXGH09] CHAN S.-M., XIAO L., GERTH J., HANRAHAN P.: Maintaining interactivity while exploring massive time series. In *IEEE Symposium on Visual Analytics Science and Technology (VAST)* (Columbus, OH, October 2009). URL: http://www.purdue.edu/discoverypark/vaccine/assets/pdfs/publications/pdf/MaintainingInteractivity.pdf. 10

[DB13] DEAN J., BARROSO L. A.: The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80. URL: http://doi.acm.org/10.1145/2408776.2408794, doi:10.1145/2408776.2408794. 4

[DG04] DEAN J., GHEMAWAT S.: MapReduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation (OSDI)* (San Francisco, CA, December 2004). URL: http://labs.google.com/papers/mapreduce.html. 10

[EF10] ELMQVIST N., FEKETE J.: Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Transactions on Visualization and Computer Graphics 16*, 3 (May 2010), 439–454. URL: https://engineering.purdue.edu/~elm/projects/hieragg/hieragg.pdf, doi:10.1109/TVCG.2009.84. 10

[FMKP14] FERNANDEZ R. C., MIGLIAVACCA M., KALYVIANAKI E., PIETZUCH P.: Making state explicit for imperative big data processing. In *USENIX Annual Technical Conference* (Philadelphia, PA, June 2014), pp. 49–60. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/castro-fernandez. 10

[FPDS12] FISHER D., POPOV I., DRUCKER S., SCHRAEFEL M.: Trust me, I'm partially right: Incremental visualization lets analysts explore large datasets faster. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (New York, NY, USA, 2012), pp. 1673–1682. URL: http://doi.acm.org/10.1145/2207676.2208294, doi:10.1145/2207676.2208294. 10

[Ham14] HAMANN O.: Eagle mode project. http://eaglemode.sourceforge.net/philosophy.html, 2014. 9

[HBB*12] HALL A., BACHMANN O., BÜSSOW R., GĂNCEANU S., NUNKESSER M.: Processing a trillion cells per mouse click. *Proc. VLDB Endow. 5*, 11 (July 2012), 1436–1446. URL:

http://dx.doi.org/10.14778/2350229.2350259, doi:10.14778/2350229.2350259. 10

[HHW97] HELLERSTEIN J. M., HAAS P. J., WANG H. J.: Online aggregation. In *ACM SIGMOD International conference on Management of data (SIGMOD)* (New York, NY, USA, 1997), pp. 171–182. URL: http://doi.acm.org/10.1145/253260.253291, doi:10.1145/253260.253291. 10

[HN13] HAUSENBLAS M., NADEAU J.: Apache Drill: Interactive ad-hoc analysis at scale. *IEEE Comput. Graph. Appl. 1*, 2 (June 2013). URL: https://www.mapr.com/sites/default/files/apache_drill_interactive_ad-hoc_query_at_scale-hausenblas_nadeau1.pdf, doi:10.1089/big.2013.0011. 10

[IVM13] IM J.-F., VILLEGAS F. G., MCGUFFIN M. J.: VisReduce: Fast and responsive incremental information visualization of large datasets. In *IEEE International Conference on Big Data* (Oct 2013), pp. 25–32. doi:http://doi.org/10.1109/BigData.2013.6691710. 6, 8, 9, 10

[JE10] JAVED W., ELMQVIST N.: Stack zooming for multi-focus interaction in time-series data visualization. In *Pacific Visualization Symposium (PacificVis), 2010 IEEE* (March 2010), pp. 33–40. doi:10.1109/PACIFICVIS.2010.5429613. 9

[KJTN14] KAMAT N., JAYACHANDRAN P., TUNGA K., NANDI A.: Distributed interactive cube exploration. In *International Conference on Data Engineering (ICDE)* (March 2014), pp. 472–483. URL: http://arnab.org/files/dice.nandi_.pdf, doi:10.1109/ICDE.2014.6816674. 10

[LJH13] LIU Z., JIANG B., HEER J.: imMens: Real-time visual querying of big data. *Computer Graphics Forum (Proc. EuroVis) 32* (2013). URL: http://vis.stanford.edu/papers/immens. 6, 10

[LKS13] LINS L., KLOSOWSKI J. T., SCHEIDEGGER C.: Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2456–2465. doi:http://doi.ieeecomputersociety.org/10.1109/TVCG.2013.179. 10

[MG82] MISRA J., GRIES D.: Finding repeated elements. *Science of Computer Programming 2* (1982), 143–152. doi:10.1016/0167-6423(82)90012-0. 7

[MGL*10] MELNIK S., GUBAREV A., LONG J. J., ROMER G., SHIVAKUMAR S., TOLTON M., VASSILAKIS T.: Dremel: Interactive analysis of web-scale datasets. *Proc. VLDB Endow. 3*, 1-2 (Sept. 2010), 330–339. URL: http://dx.doi.org/10.14778/1920841.1920886, doi:10.14778/1920841.1920886. 10

[MHK09] MAMMARELLA M., HOVSEPIAN S., KOHLER E.: Modular data storage with Anvil. In *ACM Symposium on Operating Systems Principles (SOSP)* (New York, NY, USA, 2009), pp. 147–160. URL: http://www.sigops.org/sosp/sosp09/papers/mammarella-sosp09.pdf, doi:http://doi.acm.org/10.1145/1629575.1629590. 10

[Mic] MICROSOFT CORP.: Tempe. http://research.microsoft.com/en-us/projects/tempe/. Accessed: 2014-06-01. 10

[Mut05] MUTHUKRISHNAN M.: *Data Streams: Algorithms and Applications*. Foundations and Trends in Theoretical Computer Science. Now Publishers Inc., Jan. 2005. 1

[NHM*14] NELSON J., HOLT B., MYERS B., BRIGGS P., CEZE L., KAHAN S., OSKIN M.: *Grappa: A Latency-Tolerant Runtime for Large-Scale Irregular Applications*. Tech. Rep. UW-CSE-14-02-01, University of Washington, Seattle, February 2014. 10

[ont] Ontime flights database. www.transtats.bts.gov/Fields.asp?Table_ID=236. 9

[PF93] PERLIN K., FOX D.: Pad — an alternative approach to the computer interface. In *Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* (1993). 9

[PL10] POWER R., LI J.: Piccolo: Building fast, distributed programs with partitioned tables. In *Symposium on Operating System Design and Implementation (OSDI)* (Vancouver, Canada, October 4-6 2010). URL: http://www.news.cs.nyu.edu/~jinyang/pub/power-piccolo.pdf. 10

[PSGL94] PAL SINGH J., GUPTA A., LEVOY M.: Parallel visualization algorithms: Performance and architectural implications. *Computer 27*, 7 (1994), 45–55. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=299410. 9

[RV06] RAJAGOPALAN R., VARSHNEY P.: Data-aggregation techniques in sensor networks: A survey. *IEEE Communications Surveys Tutorials 8*, 4 (2006), 48–63. doi:10.1109/COMST.2006.283821. 10

[SOHL*96] SNIR M., OTTO S. W., HUSS-LEDERMAN S., WALKER D. W., DONGARRA J.: *MPI: The complete reference*. MIT Press, Cambridge, MA, 1996. 9

[spl] Splunk: Dashboards and visualizations. http://docs.splunk.com/Documentation/Splunk/latest/Viz/Aboutthismanual. 10

[Squ07] SQUILLACOTE A.: *The ParaView guide: a parallel visualization application*. Kitware, 2007. URL: http://www.paraview.org. 9

[STH08] STOLTE C., TANG D., HANRAHAN P.: Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM 51*, 11 (2008), 75–84. URL: http://mkt.tableausoftware.com/files/Tableau-CACM-Nov-2008-Polaris-Article-by-Stolte-Tang-Har pdf. 9

[Stu09] STUDIO Z.: Data virtualization. http://www.zagstudio.com/blog/498, July 26 2009. 10

[UEPPB11] ÚLFAR ERLINGSSON, PEINADO M., PETER S., BUDIU M.: Fay: Extensible distributed tracing from kernels to clusters. In *ACM Symposium on Operating Systems Principles (SOSP)* (Cascais, Portugal, October 2011). URL: http://budiu.info/work/fay-sosp11.pdf. 10

[VBS*11] VO H., BRONSON J., SUMMA B., COMBA J., FREIRE J., HOWE B., PASCUCCI V., SILVA C.: Parallel visualization on large clusters using MapReduce. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (oct. 2011), pp. 81–88. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6092321&tag=1, doi:10.1109/LDAV.2011.6092321. 10

[XZZ*14] XIAO T., ZHANG J., ZHOU H., GUO Z., MCDIRMID S., LIN W., CHEN W., ZHOU L.: Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs. In *(ICSE SEIP) Software Engineering in Practice* (April 2014). URL: http://research.microsoft.com/apps/pubs/default.aspx?id=210324. 2

[YCZ14] YAN Y., CHEN L. J., ZHANG Z.: Error-bounded sampling for analytics on big sparse data. In *International Conference of Very Large Data Bases (VLDB)* (Hangzhou, China, September 1-5 2014). URL: http://research.microsoft.com/apps/pubs/default.aspx?id=219983. 10

[YIF*08] YU Y., ISARD M., FETTERLY D., BUDIU M., ERLINGSSON Ú., GUNDA P. K., CURREY J.: DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Symposium on Operating System Design and Implementation (OSDI)* (San Diego, CA, December 8-10 2008), p. 14. URL: http://research.microsoft.com/users/mbudiu/DryadLINQ.pdf. 2

[YTL*14] YANG F., TSCHETTER E., LÉAUTÉ X., RAY N., MERLINO G., GANGULI D.: Druid: A real-time analytical data store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, pp. 157–168. URL: http://static.druid.io/docs/druid.pdf, doi:10.1145/2588555.2595631. 10

[ZCF*10] ZAHARIA M., CHOWDHURY M., FRANKLIN M. J., SHENKER S., STOICA I.: Spark: Cluster computing with working sets. In *USENIX Workshop on Hot Topics in Cloud Computing, (HotCloud)* (Boston, MA, June 22 2010). URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets. 2, 10